A Configurable Software-based Approach for Detecting CFEs Caused by Transient Faults

Wei Liu[1*], LinLin Ci[1] and LiPing Liu[1]
[1] Computer department, Beijing Institute of Technology
Beijing China
[e-mail: Liuwei_bit@126.com, cilinlin_bit@126.com, liuliping_bit@163.com]
*Corresponding author: Wei Liu

Received July 12, 2019; revised March 4, 2021; accepted May 10, 2021; published May 31, 2021

Abstract

Transient faults occur in computation units of a processor, which can cause control flow errors (CFEs) and compromise system reliability. The software-based methods perform illegal control flow detection by inserting redundant instructions and monitoring signature. However, the existing methods not only have drawbacks in terms of performance overhead, but also lack of configurability. We propose a configurable approach CCFCA for detecting CFEs. The configurability of CCFCA is implemented by analyzing the criticality of each region and tuning the detecting granularity. For critical regions, program blocks are divided according to space-time overhead and reliability constraints, so that protection intensity can be configured flexibly. For other regions, signature detection algorithms are only used in the first basic block and last basic block. This helps to improve the fault-tolerant efficiency of the CCFCA. At the same time, CCFCA also has the function of solving confusion and instruction self-detection. Our experimental results show that CCFCA incurs only 10.61% performance overhead on average for several C benchmark program and the average undetected error rate is only 9.29%. CCFCA has high error coverage and low overhead compared with similar algorithms. This helps to meet different cost requirements and reliability requirements.

Keywords: Fault tolerance, Reliability, Control flow errors, Configurability.

http://doi.org/10.3837/tiis.2021.05.014 **ISSN : 1976-7277**

1. Introduction

Transient faults are increasing due to improving circuit integration, reducing voltage levels, increasing transistor counts and reducing noise margins [1-5]. Although it does not destroy the internal structure of the integrated circuit, it can affect the operation of the program by changing the state of the processor or the memory cell, thus endangering the reliability of the system.

The CFEs are caused by instruction address errors, opcode corruption or transforms between non-jump instruction and jump instruction that occurred at program counter, bus, and address calculation unit. The CFEs account for about 30% to 70% of the total errors [6,7].

The CFEs detection technology can be divided into hardware-based methods [8-10], mixed software-hardware methods [11-13], and software-based control flow detection technology. The former requires additional hardware components to detect CFEs.

The research methods of control flow errors caused by transient faults can be divided into hardware methods, mixed software-hardware methods and software methods.

By code redundancy, the latter assigns different static signature to each basic block and provides signature operation instructions and comparison instructions. When the program is running, a dynamic signature is calculated according to the current control flow and static signature stored in precursor basic block. Finally, the control flow is judged to be normal or not according to comparison results between dynamically generated signature and pre-stored signature. The latter has some advantages over the former, that is, software-based methods do not require auxiliary hardware, have no special requirements of the operating system, have good expansibility and are conducive to the continuous exploration and repeated experiments of the subject.

The control flow errors of inner-block is a class of errors, i.e., instructions in the basic block are not executed from beginning to end, and some of them are skipped. The error occurrence of inner-block has a low probability than inter-block.

The probability of CFEs occurring within block is proportional to the instruction number. The smaller basic block contains fewer instructions, and the larger basic block contains more instructions. In addition, the algorithm for detection also increases the number of instructions. This increases the probability that CFEs will occur within a block. Checking for errors within basic blocks is essential.

Although software methods require no additional auxiliary devices, have no special requirements on the operating system, have good expansibility, and are conducive to the continuous exploration and repeated experiments of the subject. But those bring a large amount of time and space overhead due to the large number of redundant instructions which still have a huge impact on the program performance.

Our algorithm includes the basic part and the optimization part. The basic part of the algorithm is designed to be more favorable to probe the CFEs of inner-block. For improving fault tolerance efficiency, the optimization part is designed to reduce checkpoints and reconfigure detection regions by criticality analysis and configurability of block size. Our contributions are as following:

- The algorithm overcomes the problems of high cost and low detection rate of existing detection methods in the detection of inner-block CFEs, by extracting time invariants, assigning unique signature and updates signatures at virtual edges of the control flow graph.
- Our algorithm solves the confusion problem with virtual edges.

- The algorithm implemented configurability of algorithm by analyzing the criticality of each region and tuning the detecting granularity. Configurability ensures that the number of redundant instructions decreases and tolerance efficiency increases when the application requirements are met.

The remainder is arranged as following. Section 2 gives the related works on software-based methods. Section 3 gives our methods. Section 4 analyzed the detection capabilities of CFEs. The effectiveness of the proposed method is verified by experiments in section 5 and finally conclusions is given in section 6.

2. Previous Work

The control-flow checking technology implemented in software has no special requirements on hardware and operating system but inserts some extra instructions into the normal instructions of the program. Compared to hardware or hybrid implementation technologies, software implementation technologies are low cost and flexible, requiring only software modification to meet changes in demand, and can be used directly on commercial finished devices with low cost, low power consumption and high performance.

With the goal of detecting CFEs, software-based approaches are implemented by inserting signatures. Clearly, these methods differ from each other lies in the representation of signatures and the design of detection instructions. The detection overhead and detection performance also depend on these two factors.

CFCSS [14] method assign signature based on relationship of predecessor. This algorithm gives a signature s_i to a basic block. For each basic block, signatures and XOR value D between the current basic block and predecessor are inserted in advance. The XOR value between dynamic signature and D is calculated when the program is executed into a destination block. If the result is the same as the signature of current basic block, the control flow is correct. Vice versa indicates that the control flow is wrong. These methods [15-18] and CFCSS work in the same way.

Relationship signatures for control flow checking (RSCFC) [19] encodes basic blocks in binary. These bits contain information of successors blocks. When the program runs into a basic block, the algorithm checks whether the bit corresponding to the current basic block is zero. When the representative number is zero, it means that the program has an illegal jump, and the control flow jump error is detected. Otherwise, it means that the program control flow jumps normally. This encoding can represent a limited blocks number. These methods [20-22] and RSCFC work in the same way.

ECCA [23] inserts assertions into each basic block for comparing and updating. Because of the multiplication and division operations used in assertions, this comes at the expense of increased performance overhead. DSM [24] inserts detection instructions to overcome the detection vulnerability of existing algorithms, but the cost is very high, resulting in a three-fold decrease in program performance and a four-fold increase in storage consumption. CEDA [25] has the highest fault tolerance efficiency among all known algorithms. Although the algorithm can detect all control flow errors between basic blocks with low performance overhead, it fails to solve the problem of control flow error detection within basic blocks and between processes. In addition, like other existing control flow detection algorithms, CEDA does not have configurability and self-protection ability of fault tolerance mechanism. Control flow checking at virtual edges (CFCVE) [26] inserts a virtual vertex into each edge at compile time. This together with insertion of signature updating

instructions and checking instructions into corresponding vertexes and virtual vertexes. CFCVE has some improvements in performance and overhead.

3. Methodology

Existing signature monitoring based inter-block control flow error detection algorithm not only has expensive overhead, but also lack the mechanism of self-protection. To overcome these problems, CCFCA algorithm is proposed, which is an inter-block control flow detection algorithm. The proposed algorithm assigns a unique integer as a signature for each basic block. To generate virtual edge, CCFCA inserts a virtual basic block into the jump branch according to the mapping relationship between the edge of the control flow graph and the program branch. The signature updating operation is transferred to the virtual edge by inserting signature update instructions into the virtual basic block. In order to improve fault tolerance efficiency, CCFCA divides the target program into regions and analyzes their importance. The detection instructions are inserted into nodes in important region. The detection instructions are inserted into entry and exit basic blocks in unimportant region, and the signature generation instructions are inserted into the remaining basic blocks.

3.1 Preliminaries

Here are the definitions that are relevant to our article.

Definition 1 (Basic block) Let B represent the basic block. B is an instruction set which executed sequentially, consisting of a unique entry instruction and a unique exit instruction.

Definition 2 (Control flow graph) Let $CFG = <B, E>$ denote the ordinal pair which is composed of the basic block set B and the directed edge set E.

Definition 3 (Vertex/Node) Each basic block is a vertex or Node.

Definition 4 (Virtual Vertex/Node) Virtual Vertex is essentially edge that exist between two vertices/Nodes.

Definition 5 (Internode CFEs) All CFEs occur inside the basic block.

Definition 6 (Intranode CFEs) All CFEs occur between the basic blocks.

Definition 7 (Region of code) Code region is represented by R, which is essentially a collection of basic blocks.

Definition 8 (Vulnerability of code) Let $V(R_i)$ denote the vulnerability, then it can be expressed by the formula as:

$$V(R_i)=\frac{T(R_i)}{T_{MT}(R)} \times F(R_i) \tag{1}$$

where R_i is a region of code, $T(R_i)$ is the time it takes to execute all the code in R_i, $F(R_i)$ is the execution frequency of code in region, $T_{MT}(R)$ is the average running time. The above information can be obtained from the program profile information.

Definition 9 (Importance of code) Let $I(R_i)$ be the importance of code, that is, the transient faults occur in R_i, which can cause control flow errors. It can be expressed by the formula as:

$$I(R_i) = V(R_i) \times P_{SDC}(R_i) \tag{2}$$

where $V(R_i)$ is the vulnerability of code in region R_i, $P_{SDC}(R_i)$ is the probability of silent data corruption. Those can be obtained through error injection experiments.

Definition 10 (Relative Importance of code) Let $RI(R_i)$ be the relative importance of code in R_i, where $I(R_i)$ is the importance of code in R_i, $I(R)$ is numerically the sum importance of all code regions. It can be expressed by the formula as:

$$RI(R_i) = \frac{I(R_i)}{I(R)} \tag{3}$$

3.2 Signature generation

In the previous section, the CFEs detection methods based on software lie in the representation of signatures and the design of detection instructions.

A well-designed signature approach can reduce unnecessary memory overhead and performance overhead. For example, in RSCFC algorithm, a basic block needs to occupy one bit of signature. When the number of basic blocks exceeds the machine word length, multiple registers must be used to store the same signature data, which increases the complexity of signature allocation and signature operation overhead. The signature of CCFCA algorithm consists of three parts, which are signature field, entry/exit field, and checking field respectively. The signature field is a unique binary number. Entry/exit field contains two bits to indicate that the control flow has entered or left the basic block. The entry/exit signature bit is set to 11 when the program executes inside the basic blocks and to 00 when the program executes outside the basic blocks. Its benefit for detecting instruction itself. The signature checking bit represents the parity of hamming weight in the parity checking field. If the hamming weight is odd, the checking bit is 1; otherwise, it is 0. If the machine word length is N, the maximum effective length of the signature is N-3. For example, the maximum value is 2^{61} for a 64-bit machine. This representation enhances the presentation of the signature.

3.3 Detection principle

Detection techniques based on software rely on inserting redundant instructions. CCFCA uses the global signature register (GSR) to hold the runtime signature G associated with the current node. Assume that G_i is the value of GSR when the control flow enters V_i. Then the signature function $f(G, d_i) = G \oplus d_i$ is responsible for updating the value of G at run time. If $G_i = s_i$, then the control flow is normal. If $G_i \neq s_i$, then the control flow is abnormal. Suppose there is an edge br_{sd} between V_s and V_d. That is, there is a jump between V_s and V_d, where s stands for source and d for destination. Let $d_d = s_s \oplus s_d$ be the difference of signature and is pre-stored in V_d. The value in the general signature register G was $G_s = s_s$ before the edge br_{sd} appeared. After edge br_{sd} appears then the value of the register changes to $G_d = f(G_s, d_d)$. If $G_d = s_d$, then the control flow is normal. If $G_d \neq s_d$, then the control flow is abnormal. As shown in the **Fig. 1**, CCFCA can detect abnormal jump between V_1 and V_4 based on the above principle.

However, there may be confusion between the predecessor basic block and successor basic block. As shown in the **Fig. 2**, both V_1 and V_3 can reach V_5. If the signature XOR difference is set to $d_5 = s_1 \oplus s_5$, then the value of G is $G_5 = f(G_1, d_5) = G_1 \oplus d_5 = s_5$ when br_{15} appears. If br_{35} appears, $s_3 \neq s_1$ in V_5. So the value of G becomes $G_5 = f(G_3, d_5) \neq s_5$. Given $s_3 = s_1$, then the illegal jump from node 1 to node 4 and illegal jump from node 3 to node 2 are not detected. Adjustment signature D is introduced to solve this problem.

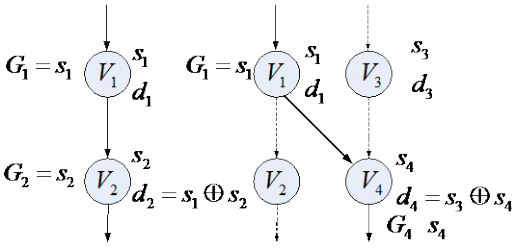


Fig. 1. Control flow detection

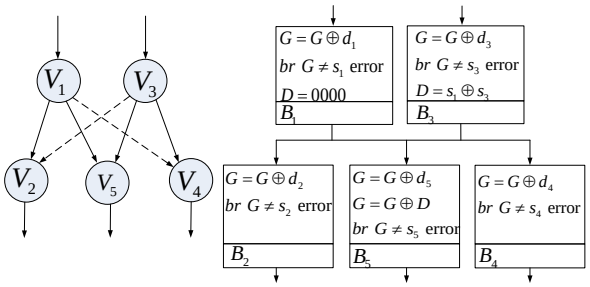


Fig. 2. Confusion can be resolved by adding D

But the confusion in **Fig. 3** cannot be solved by adjustment signature. This situation can be summed as *pred* (V_i) ≠ *pred* (V_j) and *pred* (V_i) - *pred* (V_j) ≠ ∅. If deg^- (V_i) > 1 for all *pred* (V_i), choose V_i as the base block and set D=0, then $d_5 = s_1 \oplus s_5$, $D_1 = s_1 \oplus s_1$, $D_2 = s_1 \oplus s_2$, $D_3 = s_1 \oplus s_3$, $d_6 = s_3 \oplus s_6$, $G = G_6 \neq s_6$. Legal control flow from V_3 to V_6 is misjudged as illegal control flow.

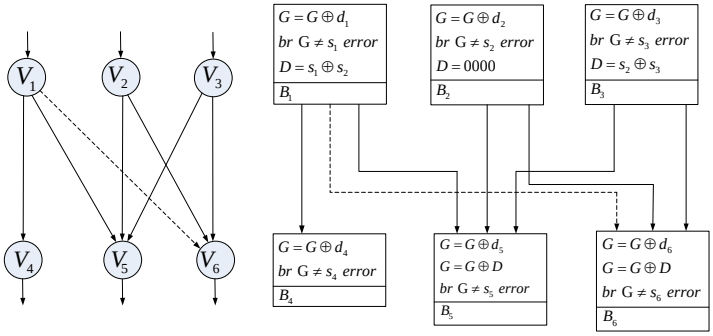


Fig. 3. Confusion can't be resolved by adding D

In the control flow error detection algorithm based on signature analysis, a legitimate branch corresponds to an edge in the edge set E. The essence of algorithm is to determine whether the edge corresponding to the current branch belongs to the legitimate edge set E. CCFCA embedded a virtual basic block in each edge of the CFG. Since the control flow of the original edge and the virtual edge are equivalent and the virtual basic block does not change the semantics of the program, the edge after inserting the virtual basic block can still be considered as an edge logically. The mechanism of updating signature with virtual base blocks more accurately expresses the relationship between the predecessor node and successor node.

CCFCA updates signature in virtual basic block by inserting signature updating instructions and detects CFEs by inserting comparison instruction in basic block. A signature comparison instruction br ($G \neq Entry$ (s_i)), updating instructions G=$G \oplus Exit$ (s_i) and $G = G \oplus Entry$ (s_j) are inserted in the virtual basic block, where $Entry(s_i)$ and $Exit(s_i)$ indicate that the entry/exit bits of signature were 0 or 1. CCFCA uses a dedicated register to store G. The value of G is s_i before V_j flows to the virtual basic block V_{ij} and the entry/exit bits of signature were 1. The first updating instruction G= $G \oplus Exit$ (s_i) is executed. The value of G is updated G = $G \oplus Entry$ (s_i) = $Entry$ (s_i) $\oplus$ $Entry$ (s_i) = 0. Then the second updating instruction G = $G \oplus Entry$ (s_j) is executed. The value of G becomes s_j and the Entry/Exit bits were 1. Next the jump instruction is executed and program is transferred to the header of basic block V_j . The

comparison instruction br (G ≠ *Entry* (s_j)) error is executed. In the absence of control-flow error, the signature value in *G* is equal to s_j, and if the value in G is not equal to s_j, then the control flow is transferred to an exception program after CFEs occurs. As shown in **Fig. 4**, V_{13} is inserted between V_1 and V_3. If V_1 jumps to V_3 along a legitimate branch then the value of *G* is updated to *Entry* (s_3). *G* = *Entry* (s_3) when the checking instruction in node V_3 is executed. At this point, you can verify that the program control flow is normal. Given V_1 jumps to V_4, the value of *G* in V_4 is not equals to *Entry* (s_4). At this point, checking instruction can detect the control flow error.

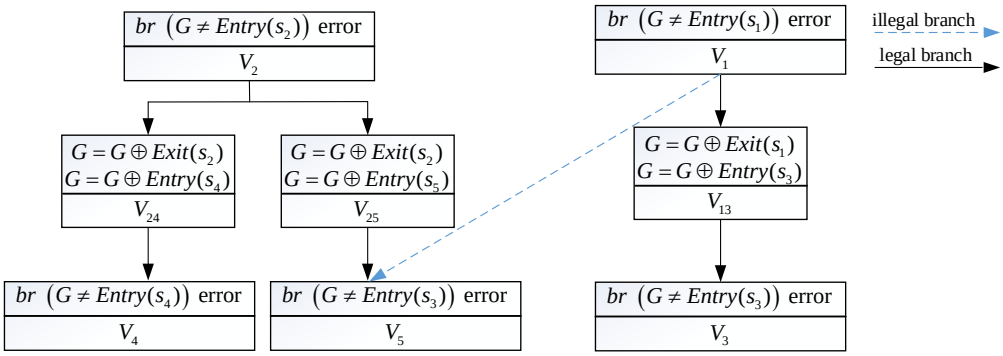


Fig. 4. An example of CCFCA

CFEs may bypass the detection mechanism by directly skipping the signature comparison instruction. Therefore, the detection mechanism should be able to detect the abnormal behavior that bypasses the detection mechanism. The Entry/Exit bits were set to 0 and 1 respectively according to the location of control flow. The status of the Entry/Exit bits is only updated within the block by inserting two instructions located in header and tail respectively. The first Entry/Exit bits instruction precedes the detection instruction. The second Entry/Exit bits instruction precedes the jump instruction. Errors skipped the first and second instructions are detected. The process of changing the status of Entry/Exit bits from 0 to 1 is driven by *EntryFactor* (s_i) and *ExitFactor* (s_i). *Transit* (s_i) indicates that the first and second bits are 1 and 0, respectively.

As shown in **Table 1**, the following conditions should be met by *EntryFactor* (s_i) and *ExitFactor* (s_i) respectively. Condition 1 ensure that the update operation only updates the target data bit in the Entry/Exit bits; Condition 2 ensure that all data bits in the signature field are unaffected; Condition 3 ensure the correctness of parity check bit.

Table 1. Condition table of *EntryFactor* (s_i) and *ExitFactor* (s_i)

Concept	Condition
EntryFactor (s_i)	(1) The first bit and the second bit are 1 and 0 respectively.
	(2) All the bits are 0 in signature field
	(3) Parity checking is 1
ExitFactor (s_i)	(1) The first bit and the second bit are 0 and 1 respectively.
	(2) All the bits are 0 in signature field
	(3) Parity checking is 1

When detecting control flow errors by redundant instructions, control flow errors may occur in redundant instructions themselves. The parity of hamming weight is consistent with

checking bit in G without SEU. There are two signature updating instructions and two signature updating instructions inside per virtual node and node respectively. The result is that these signature updating instructions may not execute in the expected order. The parity of hamming weight is consistent with checking bit due to XOR operation.

CCFCA realizes control flow error detection by redundant instruction. Suppose the number is n. The more redundant instructions, the lower the detection efficiency. Therefore, it is necessary to find a balance between efficiency and redundant instruction. If the running time of basic block B_i is T_{B_i} i=1,⋯,n. If the running time of redundant instruction C_i added to each basic block is T_{C_i} i=1,⋯,n. Then total running time of the program is $T = \sum_{i=1}^{n}(T_{B_i} + T_{C_i})$.

Suppose the average current and voltage are I and V respectively. Then the power consumption of the program is $E = P \times T$, where $P = I \times V$. The execution time and performance cost of the control flow detection algorithm are proportional to the size of the program itself and the size of redundant instruction added. A program running on a computer typically consumes most of its execution time in partial code which is relatively important code. Then the definition 7-10 related to configurability of CCFCA are given in section 3.1. Different parts of the program have different SDC error probability. If $RI(R_i) \geq \varepsilon$, the region R_i is taken as relative importance region for importance threshold ε. The remaining region is unimportant region.

CCFCA adds control flow detection instructions (including signature generation instructions, signature comparison instructions, and signature preparation instructions) to each basic block and virtual basic block in the important regions. For the relatively unimportant region, only the unique entrance and exit basic blocks of this area are configured with signature detection instructions, while the remaining basic blocks are only equipped with signature generation instructions. For relatively unimportant areas, this only increases the error detection delay of the control flow appropriately, saves time and space overhead.

4. Detection capabilities of CCFCA

The CFEs may occurs between or within basic blocks. The illegal CFEs are grouped into six types.

Type 1 Illegal branch jumps from nodes or virtual nodes to the second instruction of a node head.

Type 2 Illegal branch jumps from nodes or virtual nodes to any middle instruction of a node.

Type 3 Illegal branch jumps from node or virtual nodes to the tail instruction of a node.

Type 4 Illegal branch leaves the node from any instruction of a node head.

Type 5 Illegal branch leaves the node from any middle instruction of a node.

Type 6 Illegal branch leaves the node from any end instruction of a node.

Case 1-1 If the illegal branch jumps from the first instruction of virtual node V_i where $V_i \in pred(V_j)$, then the value in signature register is 0 before the illegal branch jump. The signature comparison instruction br (G ≠ *Transit* (s_j)) error in V_j is executed after the illegal branch jump. since the value is not equal to *Transit* (s_j) at run time. The illegal jump is detected and program running into a handling mechanism. Given illegal branch jumps from the second instruction of virtual node V_i, then signature register value is *Entry* (s_j) before the illegal branch. The signature comparison instruction br (G ≠ *Transit* (s_j)) error in V_j is executed after the illegal branch jump. Since the signature value is not equal to *Transit* (s_j) at run time. The illegal branch is detected and program runs into a handling mechanism.

Case 1-2 If the illegal branch jumps from the arbitrary instruction of node V_i where $V_i \notin pred(V_j)$, then the register values can only be one of *Entry* (s_i), *Transit* (s_i) and *Exit* (s_i). Because none of the run-time signatue values are equal to *Transit* (s_j). The detection mechanism transfers the control flow to error handling routines. If V_i is a virtual basic block and $V_k \in pred(V_i)$, then the run-time signature in the signature register before an illegal branch jump may have a value of 0 and *Entry* (s_k). The register value does not equal is *Transit* (s_j). So the llegal jump starts from the arbitrary instruction of basic block V_i can be detected. Based on the above analysis, the type 1 can be detected.

Case 2-1 If there is an illegal jump from virtual basic block (or basic block) V_i to basic block V_j, and $V_i \in pred(V_j)$. The case is similar to 1-1. The difference is that the illegal branch skips the detection instruction at the head of the node and the illegal jump is detected by the detection instruction.

Case 2-2 If there is an error from virtual node (or node) V_i to V_j where $V_i \notin pred(V_j)$. The case is similar to 1-2. The difference is that the illegal branch skips the detection instruction at the head of the node and the illegal jump is detected by the detection instruction at the end of the node. Case 2 can be detected.

Case 3-1 The proof is the same as type 2.

Case 4-1 If there is an illegal branch from virtual node (or node) V_i to node V_j where $V_j \in suc(V_i)$, then the value of register is Transit (s_i). If the illegal branch jumps to the initiation of V_j, then the $G=G \oplus$ *Exit* (s_i) and $G=G \oplus$ *Entry* (s_j) are executed. The control flow is then sequentially executed to the head of V_k which is the successor of V_j. When the second instruction of V_k is executed, the dynamic signature will be compared with *Transit* (s_k). Illegal jumps are detected when the comparison results are not equal. The scenario where an illegal branch jumps to the other two instructions is similar to the detection scenario where it jumps the first instruction.

Case 4-2 If there is an illegal branch jumps from virtual node V_i to node V_j where $V_j \notin suc(V_i)$, then the value of register is *Transit* (s_i). The instruction br (G ≠ *Transit* (s_j)) error is executed when the illegal branch jumps to the head of V_j. The value of register G updated to G=*Transit* (s_i) ⊕ *Transit* (s_j). At this point, the detection mechanism detects the control flow error. The instruction br (G ≠ *Exit* (s_j)) error is executed when the illegal branch jumps to the middle or end of V_j. The value of register G is not equal to *Exit*(s_j). At this point, the detection mechanism detects the illegal control flow. If V_j is a virtual basic block and its successor is V_k, then the instruction $G=G \oplus$ *Exit* (s_i) and the instruction $G = G \oplus$ *Entry* (s_j) are executed after the illegal branch jump to the first instruction of V_j. And the run-time signature will compared with signature *Transit* (s_k). The CFE is detected. The illegal branch jumps to the second or the third instruction is similar to the first instruction. Based on the above analysis, the type 4 can be detected.

Case 5-1 The proof is the same as type 4.

Case 6-1 If there is an illegal jump from virtual basic block (or basic block) V_i to basic block V_j , $V_j \in suc(V_i)$, $V_k \in suc(V_j)$ and the value of register is *Exit* (s_i) , then $G=G \oplus$ *Exit* (s_i) and $G = G \oplus$ *Entry* (s_j) are executed and the value of G is updated to *Entry* (s_i). The control flow enters the V_k and the value of G equals to *Entry* (s_j). Therefore, the branch is considered legal. If the illegal branch jumps to the second instruction of V_j, then the instruction $G = G \oplus$ *Entry* (s_j) is executed. And then the control flow goes into V_k.

The value of G becomes $G=Exit(s_i) \oplus Entry(s_k) \oplus EntryFactor(s_k)$. The illegal branch is detected because of the instruction br ($G \neq Transit(s_k)$) error. The illegal branch jumps to the third instruction is similar to the second instruction.

Case 6-2 If V_j is a basic block and $V_j \notin suc(V_i)$, then the value of G equals to $G=Exit(s_i) \oplus Transit(s_k)$ after the instruction br ($G \neq Transit(s_j)$) error. At this point, the illegal branch is detected. If V_j is a virtual basic block and $V_k \in suc(V_j)$, then $G=G \oplus Exit(s_i)$ and $G = G \oplus Entry(s_j)$ are executed after the illegal branch jumps to the first instruction of V_i. The G will compare with $Transit(s_k)$ after the second instruction of V_k is executed. At this point, the illegal branch is detected. The rest is similar to those described above. Based on the above analysis, the type 6 can be detected. CCFCA can detect all CFEs between basic blocks.

5. Experimental evaluation

5.1 The experiment method

The experiments used in this section are reinforcement experiment, contrast experiment and fault tolerance efficiency experiment commonly used in this research field. The experimental environment, tools and benchmark programs used in our experiments are common in such studies. The algorithms CFCSS, RSCFC, CEDA, CFCVE and BGCFC used for comparison are also representative algorithms in the latest findings.

The validation of fault-tolerant technology usually uses some benchmark programs for fault injection. This paper uses five benchmark programs including Matrix Multiplication (MM), Traveling Saleman Problem(TSP), Quick Sort(QS), Shuffle and Hanoi which are widely used in other similar algorithms.

The main technical steps in our experiments include source code compilation into assembly language, redundant code insertion, compilation and simulation. In our experiment, the evaluation is using the SimpleScalar simulator 3.0 [27] developed by Todd Austin. The SimpleScalar emulator is deployed on target machines with an ARM920T processor, 16G of SDRAM and operation system of Linux kernel 2.6.32. All experiments in this section use open source software SDCC [28] to compile programs into assembly code. The preprocessor is written by Flex++ lexical analyzer [29]. The simulator tests the program after interpreter and connector generate the code. PINFI [30] is used for error injection at the assembly level.

5.2 Performance evaluation

The core of the reinforcement experiment is the selective instruction redundancy technology, signature analysis and control flow detection technology. Without reinforcement mechanism, we test the system detection, running time, detection of mechanism, result correctness. Then with CCFCA, we test the system detection, running time, detection of mechanism, result correctness again. By comparing the change of system detection, running time, detection of mechanism, result correctness, we can see the effectiveness of CCFCA.

Firstly, a certain number of benchmark programs are selected, and the selected benchmark programs are copied into two copies. The original programs and the reinforced programs are compiled and the set of target instructions and operands for fault injection is selected. PINFI is used to implement fault injection in assembly language layer. Like other literatures, the experiment in this section is based on SEU. 5000 injections were performed. According to the detection result of CFEs, the injection results are divided into five types including SD, T, D, IR and CR. **Table 2** lists these types.

Table 2. Detection result of injection

Type	Annotation
SD (System Detection)	The fault is detected by system
T (Timeout)	The program is hung or exceeded the running time
D (Detected)	The fault is detected by detection mechanism
IR (Incorrect Result)	The fault is not detected but the output is wrong
CR (Correct Result)	The fault is not detected but the output is correct

As shown in **Table 3**, timeout failures accounted for 5.0%, 3.37%, 1.54%, 5.2%, and 7.5% of the total number of tests in each benchmark program, respectively. In the third row of the table. Faults with incorrect results accounted for 14.0%, 37.8%, 35.6%, 52.5% and 19.5% of the total number of tests in each benchmark program, respectively. The fourth row in the table shows that despite the failure, the program still runs to the end and produces the correct results. The benign failure accounted for 26.0%, 33.63%, 31.43%, 19.0% and 60.0% of the total experiments of various benchmark programs, respectively. The fifth line in the table refers to the system faults detected by the operating system, accounting for 55.0%, 46.0%, 37.0%, 43.5% and 39.0% of the total experiments of various benchmark programs, respectively. As shown in the last row of the table, the proportion of unsafe failures generated by the failure injection experiment is 19.0%, 41.17%, 37.14%, 57.7% and 27.0%, respectively, whose value is equal to the sum of the timeout faults and the result error faults. The original program did not add an additional detection mechanism during the failure injection experiment. Therefore, the D of each original program in line 6 is 0.

In addition to these, the difference between the result of the benchmark programs are determined by the type of the programs. For example, jumping intensive algorithms have small basic blocks and a lot of redundant instructions. The calculation intensive algorithms have bigger basic blocks and smaller redundant instructions than jumping intensive algorithms. The benchmark program has a high probability of producing correct results without detection mechanism. The reason is that some faults are shielded by the program itself. After analysis, it can be seen that the fault can be shielded by the program itself for two reasons :(1) when the fault is injected to the target address of the condition branch, if the jump operation does not occur, the injected fault will not affect the program results;(2) the injection failure may cause some instructions of the program to be repeated or skipped. If the instructions are repeated or skipped without changing the program semantics, the results of the program will not be affected.

After CCFCA reinforcement, timeout failures accounted for 1.0%, 2.2%, 12.0%, 1.70% and 1.75% of the total number of tests in each benchmark program, respectively. In the third row of the table. Faults with incorrect results accounted for 2.0%, 7.94%, 1.1%, 5.7% and 5.2% respectively. The fourth line adopts the algorithm in this chapter as a detection mechanism, and the faults detected account for 29.4%, 36.6%, 34.3%, 37.8% and 39.9% of the total number of experiments in each benchmark program, respectively. The correct result failures in the fifth row of the table accounted for 47.0%, 31.2%, 42.0%, 33.2% and 30.2% respectively. The failures detected by the operating system in the sixth row accounted for 20.6%, 20.6%, 19.3%, 19.7% and 21.0% respectively. The faults not secured accounts for 3.0%, 9.94%, 13.1%, 7.4% and 6.95% respectively. Its value is equal to the sum of incorrect result fault and timeout fault percentage. The average undetectable error was 8.078%.

CCFCA significantly increases the fault coverage value and enhances the fault coverage ability. CCFCA based on the mapping relation uses the virtual node to generate virtual edge, transfers the signature updating operations to a virtual edge, so accurately expresses the relationship between precursor node and successor node, simplifies the control graph algorithm

and reduces the test cost. In addition, the algorithm further carries on the key area analysis. These can further reinforce the benchmark programs without a huge number of instructions. Coverage of faults is the sum of D, T, CR, and SD. In **Fig. 5**, S and R2 represent the coverage of fault of benchmark program without and with reinforced by CCFCA respectively. The CF of MM, TSP, QS, Shuffle and Hanoi was 86%, 62.92%, 64.4%, 47.5% and 80.5% respectively before reinforcement. After the reinforcement, the CF becomes 98%, 92.6%, 98.9%, 94.3% and 94.8% respectively.

Some injection errors were not detected resulting in FNS. As show in **Fig. 6**, IR is equal to the sum of T and IR. The FNS of MM, TSP, QS, Shuffle and Hanoi was 19.0%, 41.7%, 37.14%, 57.7% and 27.0% respectively. These benchmarks have been reinforced so that FNS becomes 3.0%, 9.94%, 13.1%, 7.4% and 6.95% respectively.

Table 3. Original programs without reinforcement

	MM	**TSP**	**QS**	**Shuffle**	**Hanoi**
T (%)	5.0/1.0	3.37/2.2	1.54/12.0	5.2/1.70	7.5/1.75
IR (%)	14.0/2.0	37.8/7.94	35.6/1.1	52.5/5.7	19.5/5.2
CR (%)	26.0/47.0	33.63/31.2	31.43/42.0	19.0/33.2	60.0/30.2
SD (%)	55.0/20.6	46.0/20.6	37.0/19.3	43.5/19.7	39.0/21.0
D (%)	0/29.4	0/36.6	0/34.3	0/37.8	0/39.9
Total (%)	100	100	100	100	100
Fault not secured (%)	19.0/3.0	41.17/10.14	37.14/13.1	57.7/7.4	27.0/6.95

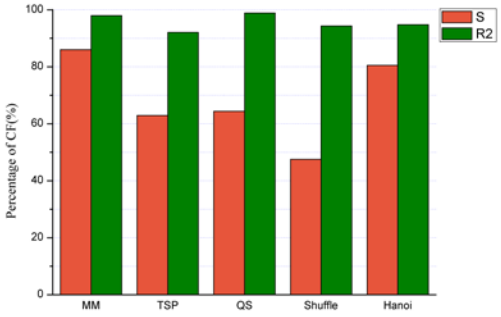


Fig. 5. The comparison of CF

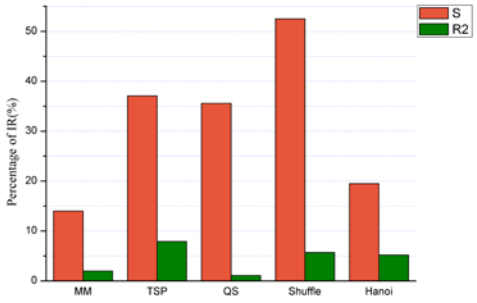


Fig. 6. The comparison of IR

The next experiment aims to compare multiple detection algorithms BGCFC, CFCSS, CEDA, CFCVE, RSCFC and CCFCA in terms of T, IR, CR, SD and D. In this part of the experiment, data comparison, histogram display and analysis of the characteristics of various algorithms are used to make a comprehensive comparison between algorithms.

Fig. 7 shows the detection performance of algorithms BGCFC, CFCSS, CEDA, CFCVE, RSCFC and CCFCA in terms of T, TR, D, CR and SD. S, B, C1, C2, C3, R1 and R2 in each set of bar respectively represent the original , BGCFC, CFCSS, CEDA, CFCVE, RSCFC and CCFCA. The CF of original program, BGCFC, CFCSS, CEDA, CFEVE, RSCFC and CCFCA is 68.264%, 92.08%, 87.6%, 86.94%, 91.688%, 91.678% and 95.615% respectively.

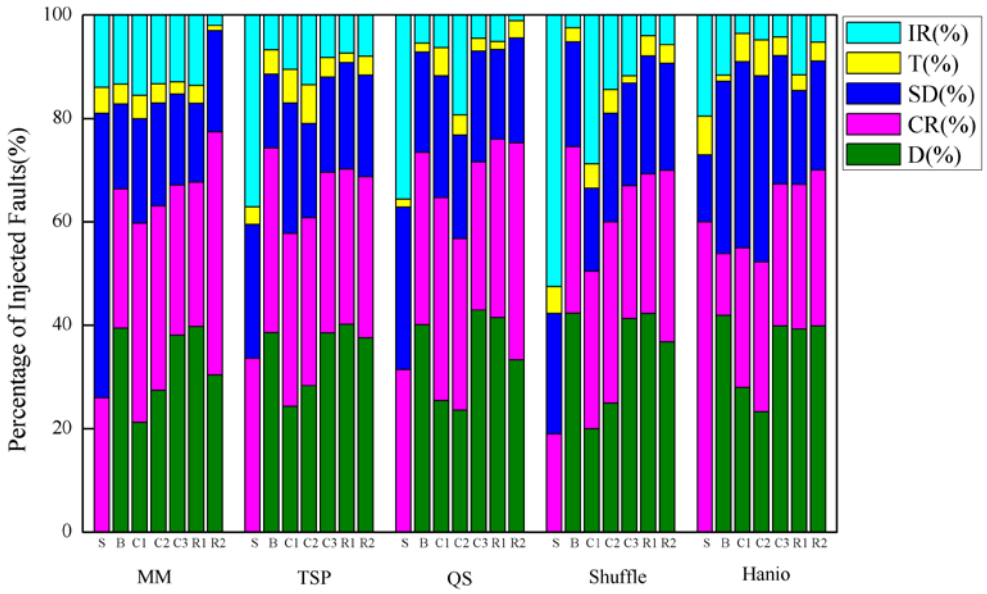


Fig. 7. Detection performance comparison of five algorithms

The T of original program, BGCFC, CFCSS, CEDA, CFEVE, RSCFC and CCFCA is 4.522%, 2.834%, 5.3%, 5.32%, 2.734%, 2.76% and 3.052% respectively. The SD of original program, BGCFC, CFCSS, CEDA, CFEVE, RSCFC and CCFCA is 29.73%, 20.714%, 24.2%, 23.02%, 20.412%, 18.808% and 20.24% respectively. The CR of original program, BGCFC, CFCSS, CEDA, CFEVE, RSCFC and CCFCA is 34.012%, 28.022%, 33.74%, 33.08%, 28.4%, 29.494% and 36.72% respectively. The D of original program, BGCFC, CFCSS, CEDA, CFEVE, RSCFC and CCFCA is 0, 40.51%, 23.82%, 25.52%, 48.148%, 40.616% and 35.6% respectively. Even without detection algorithms, the failure coverage of the original program averaged 68.2%. The reason is that part of the fault is shielded by the program itself, the jump does not occur, instructions are repeated or skipped without changing the program semantics.

CFCVE algorithm adopts virtual node to represent branch jump relation instead of writing branch jump relation into signature and effectively solves the non-detection zone and performance overhead problem. However, the self-protection mechanism of CFCVE results in additional instruction overhead. Each basic block of CEDA is assigned an entry signature and an exist signature. In order to overcome the conflict of the sector nodes, CEDA designs the same exit signature for each precursor node. But it greatly increases the complexity of signature allocation. CEDA algorithm overcomes the detection vulnerability of CFCSS algorithm, it does not have the ability to detect control flow within blocks and between processes. When the benchmark program contains a basic block with a large instruction size, the IR and FNS will increase. RSCFC algorithm does not have the ability to detect control flow errors in blocks. RSCFC encodes the jump relation between the basic blocks into the signatures and assigns the position information signature to each basic block. At runtime, RSCFC performs logic operation on dynamic signature and position information signature to detection. RSCFC solves the sector conflict well with high error coverage. However, coding the jumping relation between the basic blocks into the signature makes the signature length exceed the machine byte length. To solve this problem, block coding is required for basic blocks, which increases the algorithm

complexity and detection cost. The node is divided into two sub-nodes to convert the equivalence, so that the control flow error between the sub-blocks, to achieve the purpose of reducing the error type of the control flow. BGCFC has time complexity O(N!). When the number of nodes is large and the jump is complex. The BGCFC algorithm overcomes the problem of excessive overhead of RSCFC but inherits the advantage of high CF. The basic principle of CCFCA algorithm combines the advantages of CFCSS and CFCVE by region importance analysis and overcomes the confusion by virtual basic block. Since the condition branch does not occur, the IR of CFCSS and CCFCA have been significantly reduced.

The number of redundant instructions is related to the program type. Branch jump intensive program has small basic block and jumps frequently. Operation intensive program has large node. The instructions inserted in each node is basically the same, the more basic blocks obtained, the more instruction inserted.

As show in **Fig. 8**, BGCFC divides each node into two sub-nodes. This avoids the case where the starting node of a legitimate jump branch is the same basic block as the destination node. BGCFC algorithm assigns consecutive integer signature for predecessor of each basic block. When the number of predecessor nodes is 1, two detection instructions are inserted into the basic block; when the number of predecessor nodes is greater than 1, up to 8 detection instructions are inserted into the basic block. RSCFC assigns jump relation and location information to each node of the benchmark programs. When the node number is larger than the machine expression number, multiple registers are introduced to store the same signature data. Therefore, RSCFC assigns at least three special registers and seven detection instructions to each basic block. RSCFC algorithm and BGCFC algorithm have the same performance overhead when the node number is less than the machine expression number. RSCFC algorithm has a slightly lower performance overhead than BGCFC algorithm when the node number is larger than the machine expression number. For example, the additional overhead of BGCFC algorithm in QS, TSP and Hanoi programs are 22.3%, 21.5% and 43.7% respectively, which are lower than the 32.7%, 33.2% and 42.31% of RSCFC algorithm. CFCVE algorithm expresses branch jump relation through virtual edge and inserts 4 instructions in each basic block and 3 instructions in each virtual basic block to complete the detection. On average, the algorithm needs to insert 5.5 instructions on average to complete the detection task. CFCVE has less performance overhead than BGCFC in computationally intensive programs. For example, the BGCFC algorithm in the MM program has an additional 17.4% performance overhead, and the CFCVE algorithm has an additional 16.0% performance overhead. BGCFC algorithm in Shuffle program added an additional 43.21% of the performance overhead, CFCVE algorithm added an additional 33.14% of the performance overhead. CCFCA algorithm divides the importance of the code area and inserts 3-5 instructions on average. CEDA algorithm inserts 4 instructions in each node. However, the insertion number of CCFCA algorithm is still less than CFCSS algorithm. Therefore, the performance overhead of CCFCA algorithm is lower than that of CFCSS algorithm and CEDA algorithm. For example, CFCSS, CEDA, and CCFCA algorithms have an additional performance overhead of 13.7%, 14.0%, and 10.5%, respectively, in the program MM and Shuffle. Additional performance overhead in the Shuffle was 23.0%, 24.1%, and 20.5% respectively.

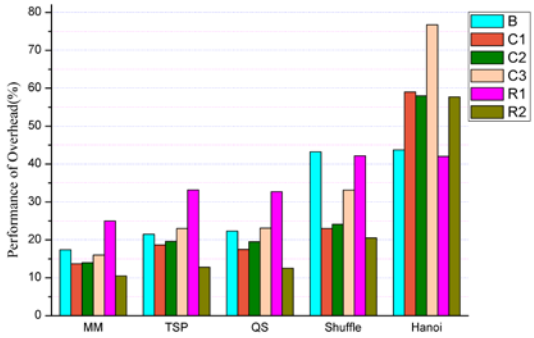


Fig. 8. Overhead comparison of five algorithms

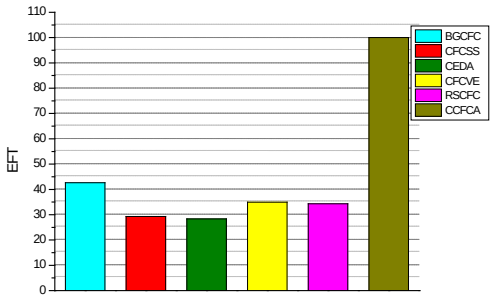


Fig. 9. Comparison of EFT

Efficiency of fault tolerance (EFT) combines the fault coverage (CF) with performance Overhead (PO). It can be used to measure control flow detection methods. The formula of EFT is expressed as follows:

$$EFT = \frac{1}{(1-CF)\times PO}, PO = \frac{T_a}{T_b} \times 100\% - 1 \quad (4)$$

Here T_a and T_b respectively represent the time executed the after reinforcement and the time taken to execute the program before reinforcement. In MM, TSP, QS, Shuffle and Hanoi, the average performance overhead of BGCFC, CFCSS, CEDA, CFCVE, RSCFC and CCFCA was 29.624%, 26.38%, 27.04%, 34.398%, 35.0212 and 22.8%, respectively. Therefore, as shown in **Fig. 9**, the fault tolerance efficiency of the six detection algorithms BGCFC, CFCSS, CEDA, CFCVE, RSCFC and CCFCA is 42.6217%, 29.2948%, 28.3171%, 34.9752%, 34.3116% and 100.02% respectively. Obviously, CCFCA algorithm proposed in this paper has the largest value of fault tolerance efficiency and the best comprehensive detection performance, which is higher than other control flow detection algorithms.

6. Conclusions and Future Research

This paper presents CCFCA algorithm. The algorithm is configurable by an assessment of the importance of code. For critical regions, program blocks are divided according to space-time overhead and reliability constraints, so that protection intensity can be configured flexibly. For other regions, signature detection algorithms are used only in the first and last nodes of the region. It is benefit for improving the fault-tolerant efficiency of the CCFCA. At the same time, CCFCA also has the function of solving confusion and instruction self-detection. Our experimental results show that CCFCA has high fault tolerance and low overhead. This helps to meet different cost requirements and reliability requirements.

Further work is to eliminate the inaccessible path and generate the control flow graph accurately. It helps optimize CCFCA to reduce redundancy.

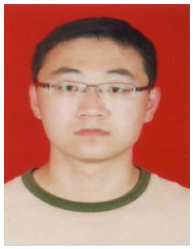
Acknowledgment

Our article receives the National Natural Science Foundation of China under grant No. 61370134, the National High Technology Research and Development Program of China (863 Program) under grant No. 2013AA013901.

References

[1] Wang P, et al., "X-ray and proton radiation effects on 40 nm CMOS physically unclonable function devices," *IEEE Transactions on Nuclear Science*, 65(8), 1519-1524, 2018. Article (CrossRef Link)
[2] Musenich R, Calvelli V, Giraudo M, et al., "The limits of space radiation magnetic shielding: an updated analysis," *IEEE Transactions on Applied Superconductivity*, 28(3), 1-5, 2018. Article (CrossRef Link)
[3] Chen W, Varanasi N, Pouget V, et al., "Impact of VCO topology on SET induced frequency response," *IEEE Transactions on Nuclear Science*, 54(6), 2500-2505, 2007. Article (CrossRef Link)
[4] Dicello J F, Paciotti M, Schillaci M E, "An estimate of error rates in integrated circuits at aircraft altitudes and at sea level," *Nuclear Instruments and Methods in Physics Research Section B: Beam Interactions with Materials and Atoms*, 40(2), 1295-1299, 1989.
[5] Wang H B, Kauppila J S, Lilja K, et al., "Evaluation of SEU performance of 28-nm FDSOI flip-flop designs," *IEEE Transactions on Nuclear Science*, 64(1), 367-373, 2017. Article (CrossRef Link)
[6] Ohlsson J, Rimen M, Gunneflo U, "A study of the effects of transient fault injection into a 32-bit RISC with built-in watchdog," in *Proc. of IEEE International Symposium on Fault-Tolerant Computing*, 316-325, 1992. Article (CrossRef Link)
[7] Schuette M A, Shen J P, "Processor Control Flow Monitoring Using Signatured Instruction Streams," *IEEE Transactions on Computers*, C-36(3), 264-276, 1987. Article (CrossRef Link)
[8] Parra L, Lindoso A, Portela M, et al., "Efficient mitigation of data and control flow errors in microprocessors," in *Proc. of European Conference on Radiation & Its Effects on Components & Systems*, pp. 1-4, 2013. Article (CrossRef Link)
[9] Namjoo M, McCluskey E J, "Watchdog processors and capability checking," in *Proc. of Twenty-Fifth International Symposium on Fault-Tolerant Computing*, 1995. Article (CrossRef Link)
[10] Azambuja, Altieri M, Becker J, et al., "HETA: hybrid error-detection technique using assertions," *IEEE Transactions on Nuclear Science*, 60(4), 2805-2812, 2013. Article (CrossRef Link)
[11] Ahmed R, El Sayed M, Gadsden S A, et al., "Automotive internal-combustion-engine fault detection and classification using artificial neural network techniques," *IEEE Transactions on Vehicular Technology*, 64(1), 21-33, 2015. Article (CrossRef Link)
[12] Zheng B, Gao Y, Zhu Q, et al., "Analysis and optimization of soft error tolerance strategies for real-time systems," in *Proc. of International Conference on Hardware/software Codesign and System Synthesis*, 55-64, 2015. Article (CrossRef Link)
[13] Mitra D, Ghoshal S, Rahaman H, et al., "On-line error detection in digital microfluidic biochips," in *Proc. of the 21th Test Symposium*, 332-337, 2012. Article (CrossRef Link)
[14] Oh N, Shirvani P P, Mccluskey E J, "Control-flow checking by software signatures," *IEEE Transactions on Reliability*, 51(1), 111-122, 2002. Article (CrossRef Link)
[15] Boroomandnezhad T, Azgomi M A, "An efficient control-flow checking technique for the detection of soft-errors in embedded software," *Computers and Electrical Engineering*, 39(4), 1320-1332, 2013. Article (CrossRef Link)
[16] Wang H, Wang H, Jin Z, "Bipartite graph-based control flow checking for COTS-based small satellites," *Chinese Journal of Aeronautics*, 28(3), 883-893, 2015. Article (CrossRef Link)
[17] Mohamed A, Zulkernine M, "A control flow representation for component-based software reliability analysis," in *Proc. of the 6th IEEE International Conference on Software Security and Reliability*, 1-10, 2012. Article (CrossRef Link)

[18] Rajabpour N, Sedaghat Y. A, "software-based error detection technique for monitoring the program execution of RTUs in SCADA," *Computer Safety, Reliability, and Security, Springer*, 2015. Article (CrossRef Link)
[19] Li A, Hong B, "On-line control flow error detection using relationship signatures among basic blocks," *Computers & Electrical Engineering*, 36(1), 132-141, 2010. Article (CrossRef Link)
[20] Asghari S A, Abdi A, Taheri H, et al., "SEDSR: Soft Error Detection Using Software Redundancy," *Journal of Software Engineering & Applications*, 5(9), 664-670, 2012. Article (CrossRef Link)
[21] Asghari S A, Taheri H, Pedram H, et al., "Software-based control flow checking against transient faults in industrial environments," *IEEE Transactions on Industrial Informatics*, 10(1), 481-490, 2013. Article (CrossRef Link)
[22] Pandiyan G, Krishnakumari P, "An efficient software defect prediction model using optimized tabu search branch and bound procedure," *Metropolitan Museum of Art Bulletin*, 10(1), 73-79, 2015.
[23] Alkhalifa, Z, Nair, V.S.S, Krishnamurthy, N, et al., "Design and evaluation of system-level checks for on-line control flow error detection," *IEEE Transactions on Parallel & Distributed Systems*, 10(6), 627-641, 1999.
[24] Nicolescu B, Savaria Y, Velazco R, "Software detection mechanisms providing full coverage against single bit-flip faults," *IEEE Transactions on Nuclear Science*, 51(6), 3510-3518, 2004. Article (CrossRef Link)
[25]Vemu R, Abraham J, "CEDA: Control-Flow Error Detection Using Assertions," *IEEE Transactions on Computers*, 60(9), 1233-1245, 2011. Article (CrossRefLink)
[26] Liping LIU, Linlin CI, Wei LIU, et al., "Control Flow Checking at Virtual Edges," *KSII Transactions on Internet and Information Systems*, 11(1), 396-413, 2017. Article (CrossRefLink)
[27] T. Austin, E. Larson and D. Ernst, "SimpleScalar: an infrastructure for computer system modeling," *Computer*, 35(2), 59-67, 2002. Article (CrossRefLink)
[28] Baldassin A, Centoducatte P, Rigo S, et al., "An open-source binary utility generator," *ACM Transactions on Design Automation of Electronic Systems*, 13(2), 1-17, 2008. Article (CrossRefLink)
[29] Kai Zhang, Junchang Wang, Bei Hua, "Building High-Performance Application Protocol Parsers on Multi-core Architectures," in *Proc. of* 2011 IEEE 17th International Conference on Parallel and Distributed Systems, 2012. Article (CrossRefLink)
[30] Luk C K, Cohn R, Muth R, et al., "Pin: building customized program analysis tools with dynamic instrumentation," *ACM Sigplan Notices*, 40(6), 190-200, 2005. Article (CrossRefLink)



Wei Liu received the B.S. degree and M.S. degree from North University of China from Dept. of Computer Science, China, in 2008 and 2011, respectively. Currently, he is studying for his PhD Degree at Computer Science in Beijing Institute of Technology. His current research interests include Secure Wireless Sensor Networks, Pattern Recognition, and Trusted Computing.

LinLin Ci received the B.S. degree in the Dept. of Computer Science from Beijing Institute of Technology, China, in 1976; he received the M.S. degree in the Dept. of Computer Science from Northwestern Polytechnical University, China, in 1985. Currently, he is professor and doctoral supervisor in computer application. His research areas include Secure Wireless Sensor Networks, Pattern Recognition, and Trusted Computing.

LiPing Liu received the B.S. degree and M.S. degree in the Dept. of Computer Science from North University of China, China, in 2008 and 2011, respectively. Currently, he is studying for his PhD Degree at Computer Science in Beijing Institute of Technology. His current research interests include Secure Wireless Sensor Networks